Comput Optim Appl
DOI 10.1007/s10589-007-9092-2

Design and implementation of a massively parallel version of DIRECT

Jian He · Alex Verstak · Layne T. Watson · Masha Sosonkina

Received: 20 March 2006 / Revised: 7 August 2006

Abstract This paper describes several massively parallel implementations for a global search algorithm DIRECT. Two parallel schemes take different approaches to address DIRECT's design challenges imposed by memory requirements and data dependency. Three design aspects in topology, data structures, and task allocation are compared in detail. The goal is to analytically investigate the strengths and weaknesses of these parallel schemes, identify several key sources of inefficiency, and experimentally evaluate a number of improvements in the latest parallel DIRECT implementation. The performance studies demonstrate improved data structure efficiency and load balancing on a 2200 processor cluster.

Keywords Data structures · DIRECT · Global search · Load balancing · Task allocation

1 Introduction

The availability of compute power on large scale parallel systems offers both potential and challenges for solving high dimensional scientific optimization problems using global search algorithms. Many of these algorithms have large memory requirements and strong data dependency that degrade the program scalability and parallel

J. He · A. Verstak
Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

L.T. Watson (✉)
Departments of Computer Science and Mathematics, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA
e-mail: ltw@vt.edu

M. Sosonkina
Ames Laboratory, Iowa State University, Ames, IA 50011, USA

efficiency as more and more processors join the workforce. The global search algorithm DIRECT (Dividing RECTangle) by Jones et al. [16] is one such algorithm. Several research projects [13, 14, 22] address its parallel design issues on large systems. Baker et al. [22] discuss the performance of several load balancing strategies for a fully distributed version of DIRECT, which solved a 28-dimensional problem on a 256 processor supercomputer. He et al. in [13, 14] tested two different parallel schemes with various problem scales on a 200 node Opteron cluster of workstations. The intent here is to present the history of these evolving parallel DIRECT implementations. Finally, the current improved parallel scheme is explored analytically and experimentally on System X, a 2200 processor Apple Xserve G5 cluster at Virginia Polytechnic Institute and State University. The performance studies focus on data structure efficiency and load balancing.

In the past decade, DIRECT has been successfully applied to many modern large scale multidisciplinary engineering problems [2, 3, 11]. Recently, DIRECT has been used in global nonlinear parameter estimation problems in systems biology [17]. However, unnecessary overhead and complexity caused by inefficient implementation inside other software packages (e.g., Matlab) may obscure DIRECT's advanced features. Some computational biologists are attracted by its unique strategy of balancing global and local search, its selection rules for potentially optimal regions according to a Lipschitz condition, and its easy-to-use black box interface. Like other global optimization approaches of [6, 8], DIRECT is being challenged by high-dimensional ($\geq$50) problems including nonlinear models for parameter estimation. The present work applies DIRECT to a 143-parameter estimation problem for a budding yeast cell cycle model [20].

As the scale of both computational problems and clusters of workstations has grown, parallel optimization algorithms have become a very active research area. However, the nature of the DIRECT algorithm presents both potential benefits and difficulties for a sophisticated and efficient parallel implementation. Gablonsky [7] and Baker et al. [22] are among the few parallel DIRECT implementations known in the public domain. In [7], Gablonsky adopts a master-slave paradigm to parallelize the function evaluations, but little discussion is given to the issue of parallel performance and potential problems, such as load balancing and interprocessor communication, both of which raise many challenging design issues. A major contribution in [22] is a distributed control version equipped with dynamic load balancing strategies. Nevertheless, that work did not fully address other design issues such as a single starting point and a strong data dependency (each processor requires logical access to the entire data structure).

At a high level, DIRECT performs two tasks—maintaining data structures that drive its selection of regions to explore and evaluating the objective function at chosen points. One of the limitations of DIRECT lies in the fast growth of its intermediate data. Jones [15] comments that DIRECT suffers from the curse of dimensionality that limits it to low dimensional problems ($<$20). Figure 1 shows the growth of the number of search subregions ("boxes") for a standard test problem with dimensions $N = 10$, 50, 100, and 150. The amount of data grows rapidly as the DIRECT search proceeds in iterations, especially for high dimensional problems. This dictates techniques to reduce the size of data structures, thus the number of machines required

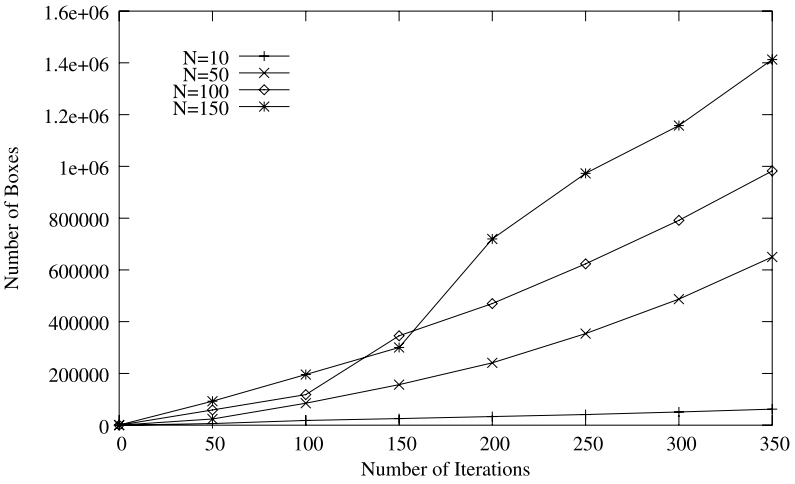


Fig. 1 The growth of number of boxes for a test problem with dimensions $N = 10$, 50, 100, and 150

to hold the distributed data in memory. The second task of evaluating the objective function at sample points presents its own challenges. The selection of sample points in the current iteration depends on all the points that have been sampled previously. Empirically, the inherent inefficiency of this necessary point sampling has a great impact on load balancing. Several strategies proposed in the parallel implementation of DIRECT are explored analytically and experimentally.

The paper is organized as follows. Section 2 begins with an overview of the DIRECT algorithm and its parallel design challenges. Section 3 presents two parallel schemes of DIRECT and discusses the evolution of the parallelization process. Performance studies for the latest parallel DIRECT version are presented in Sect. 4.

2 Algorithm overview and design challenges

DIRECT finds the global minimum of an objective function $f(x)$ inside an N-dimensional design space $\ell \leq x \leq u$. Each iteration of DIRECT consists of the following three main steps.

1. SELECTION identifies a set S of "potentially optimal" boxes that are subregions inside the design domain with dimension N. A box is potentially optimal if, for some Lipschitz constant, the objective function value at its center is potentially smaller than that in any other box (a formal definition of potential optimality can be found in [16]).
2. SAMPLING evaluates new points sampled around the centers of all "potentially optimal" boxes in S along their longest dimensions.
3. DIVISION subdivides "potentially optimal" boxes in S based on the function values at the newly sampled points.

These three steps repeat until the stopping condition is satisfied. Stopping conditions are important, for both theoretical convergence and practical considerations.

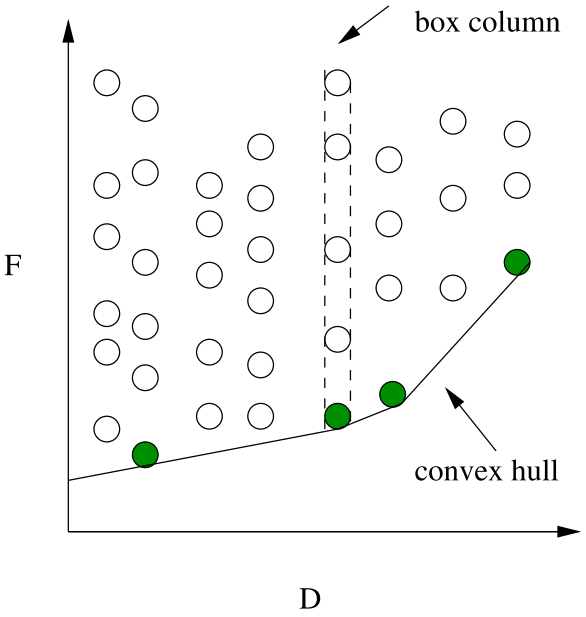


Fig. 2 An example scatter plot of boxes. The F-axis is function values, and the D-axis is box diameters

The original DIRECT algorithm had only a single stopping criterion (iteration limit), and others were proposed and studied by He et al. [12]. The stopping conditions are especially important for parallel implementations.

Initially, only one box exists in the system. As the search progresses, more boxes are generated, illustrated by the scatter plot shown in Fig. 2 in which each circle represents a box. The sizes of boxes increase along the D-axis (diameter) and the function values at box centers increase along the F-axis (function). All the boxes with the same diameter belong to a "box column". Reference [16] proves that all potentially optimal boxes in S are on the lower right convex hull of the scatter plot points in Fig. 2. Here, $\epsilon = 0$ in the definition of "potentially optimal" in [16]. Call these boxes "convex hull boxes" and the boxes with the lowest function values in each box column "lowest boxes".

In the DIVISION step, multiple new boxes are generated for each convex hull box. The multiple function evaluation tasks at each iteration give rise to a natural functional parallelism used in [7, 22]. This is especially beneficial for expensive objective functions, since the communication cost of distributing evaluation tasks to multiple processors is negligible compared to the computational cost. On the other hand, a few design challenges are also observed here. First, the algorithm starts with one box, which produces simply one evaluation task for all the acquired processors. With a single starting point, load balancing is always an issue at an early stage, even though the situation will be improved as the search progresses by subdividing the domain and generating multiple evaluation tasks. When a large number of processors is used, the load balancing issue is more critical for low dimensional problems ($<$20), which subdivide fewer boxes at every iteration than high dimensional problems. For iterations that generate fewer new boxes, a load imbalance occurs with some processors sitting idle. Second, the number of boxes subdivided at each iteration is unpredictable

depending on the result of identifying convex hull boxes. For high dimensional problems, the number of boxes grows more rapidly (as shown in Fig. 1), challenging data structure expandability and memory capacity. Therefore, data parallelism is considered here as a remedy, whereby data is distributed across multiple machines. Third, a strong data dependency that exists throughout the algorithm steps lessens program concurrency, thus degrading parallel scalability. Efficient task and data distribution strategies are demanded here for a scalable parallel DIRECT implementation, especially for large scale systems that host hundreds and thousands of processors.

3 Parallel schemes

All the observations above engendered a combined functional and data parallelization in two parallel implementations of the DIRECT algorithm, called pDIRECT_I and pDIRECT_II in what follows.

The algorithm operations can be partitioned into three functional components: domain decomposition, box subdivision, and function evaluation as shown in Fig. 3. Domain decomposition is an optional component that transforms the single start DIRECT into a multistart algorithm. Moreover, the resulting multiple subdomains are optimized independently, so that the objective function value may be reduced faster [13]. The domain decomposition is accomplished in the following two phases.

1. The longest dimension of the N-dimensional domain is subdivided into $s = \sqrt{m}$ partitions, where m is the desired number of subdomains.
2. Inside each partition above, the longest dimension is subdivided into s parts.

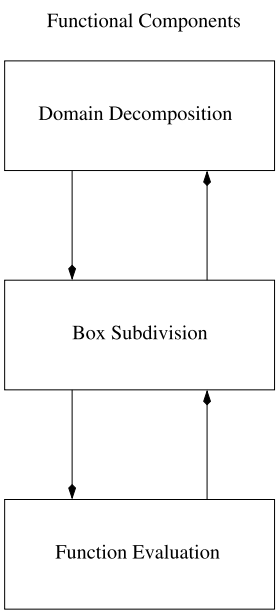


Fig. 3 Three functional components

As the second component, box subdivision applies data parallelism that spreads data across multiple processors collaborating on the SELECTION and DIVISION steps. Lastly, the function evaluation component uses the classical master-slave paradigm that distributes evaluation tasks to multiple processors during SAMPLING.

To store the unpredictable number of boxes, both pDIRECT_I and pDIRECT_II reuse the set of dynamic data structures presented in [12]. In addition, a few techniques are developed for pDIRECT_II to reduce local memory storage and network traffic. To distribute data and computation, pDIRECT_I combines a shared memory model (for box subdivision) with a message passing model (between box subdivision and function evaluation), and dynamically spawns processes for these two components. The data is distributed through the global data structures in the shared memory and computational tasks are distributed via messaging. This mixed paradigm improves data distribution efficiency compared to the pure functional parallel versions in [7–22]. However, it has its own shortcomings in program portability, processor utilization, load balancing, and termination efficiency. Therefore, the second version pDIRECT_II was developed to address these inefficiencies with a pure message passing model and more dynamic features in data structures, task allocation, and the termination process. Performance results prove that pDIRECT_II is effective for solving complex design optimization problems on modern large scale parallel systems. The following two sections first present pDIRECT_I and its design drawbacks, and then discuss the considerations leading to the improved version pDIRECT_II.

3.1 pDIRECT_I

The parallel scheme of pDIRECT_I consists of three levels as shown in Fig. 4, each level addressing one of the functional components in Fig. 3.

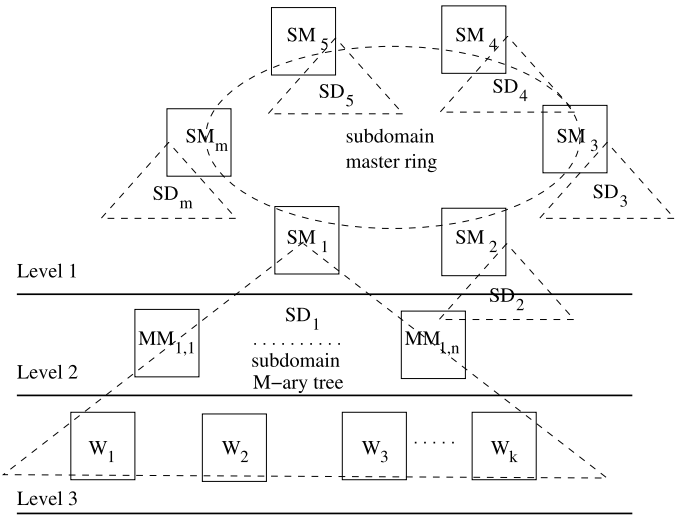


Fig. 4 The parallel scheme of pDIRECT_I

3.1.1 Topology

The processes at Level 1 form a logical subdomain master ring. The entire design domain is decomposed into multiple nonoverlapping subdomains. Each process SM_i (subdomain master i, $i = 1, \ldots, m$) on the ring starts the DIRECT search at the center of a chosen subdomain i. SM_i detects the stopping conditions, merges the results, and controls the termination at the end. SM_i is spawned at run time by MPI and joins the logical ring formed for SM processes. Level 1 uses a ring topology, because it fits the equal relationship among subdomains and represents the dependency of the stopping condition of each subdomain on other subdomains. The overall termination condition is when all subdomains have satisfied the specified search stopping criteria. In other words, a subdomain will be kept active until all search activities in other subdomains are done. This rule aims at reducing processor idleness when subdomains generate different amounts of computation. The drawback is that the stopping condition (i.e., maximum number of iterations) becomes a lower bound on the computational cost instead of an exact limit. Furthermore, the termination process is controlled by a token T passed as described in the following.

1. SM_1 issues T and passes it around the ring.
2. After the local stopping criteria are met, each SM_i checks if T has arrived at each iteration. If not, DIRECT proceeds. If yes, T is passed along in the ring.
3. After T is passed back to SM_1, a termination message is sent to all SM_i.
4. SM_1 collects the final results from all SM_i.

This process decentralizes the termination control, thus avoiding the bottleneck at SM_1 when the number of subdomains m is large. On the other hand, there are a few disadvantages of using the ring. First, the communication latency on a ring is higher than on some other topologies, such as a star or a tree. Second, the lower bound stopping condition can not provide users accurate estimates of computational cost.

Below Level 1, Level 2 uses GPSHMEM [21] to establish a global addressing space to access the data for SELECTION. This globally shared data structure corresponds to a work pool paradigm [9] that dynamically adjusts box subdivision workload among mini subdomain master (MM) processes at Level 2. GPSHMEM is a very suitable programming model for this case since it provides programming primitives for one-sided communications in a distributed memory environment. Similar in spirit to the Global Arrays [19] communication layer, GPSHMEM is a light-weight open-source library extending the useful SHMEM paradigm [5] from Cray supercomputers to general distributed memory environments. It is different from shared memory programming models, such as OpenMP [4], which are commonly used in shared memory SMP environments and regarded for their automatic parallelization capabilities via compiler directives inserted in serial code.

Between Levels 2 and 3, a master-slave paradigm is used for distributing function evaluation tasks. Both Levels 2 and 3 take advantage of dynamic process management in MPI-2 [10] so that processors are assigned to these two levels at run time with approximately $(p - m)/m$ processors available for each subdomain (out of p total processors). In Fig. 4, a $\lfloor M \rfloor$-ary tree structure is rooted at each SM process, where

$$M = \sqrt{\frac{p-m}{m}}.$$

Each *SM* process dynamically spawns $n = \lfloor M \rfloor$ mini subdomain master (*MM*) processes at Level 2 for box subdivision tasks. Similarly, each *MM* process spawns $\lfloor k \rfloor$ or $\lceil k \rceil$ worker processes for function evaluation tasks, where

$$k = \frac{p - m(1 + \lfloor M \rfloor)}{m \lfloor M \rfloor}.$$

To form the $\lfloor M \rfloor$-ary tree of processes, pDIRECT_I requires that the total number of processes $P \geq 16$. If the number of available processors $p \geq 16$, then $P = p$. Otherwise, P is set at 16, so that multiple processes may run on the same physical processor. Pseudocode 1 shows the interactions between $MM_{i,1}$ and $MM_{i,j}$ $(j = 2, \ldots, n)$ in subdomain i $(i = 1, \ldots, m)$ managed by SM_i.

Pseudocode 1

```
done := FALSE (the search status)
MM_{i,1} receives DIRECT parameters (problem size N, domain D,
    and stopping conditions C_stop) from SM_i
broadcast DIRECT parameters to MM_{i,j}
do
  if (MM_{i,1}) then
    if (done = FALSE) then
      run one DIRECT iteration and merge intermediate results
        if (C_stop satisfied) then
          done := TRUE
          send done to SM_i
        end if
        cycle
    else
      receive a message from SM_i
      if (not a termination message) then
        send a handshaking message to SM_i
        broadcast a message to keep MM_{i,j} working
        run one DIRECT iteration and merge intermediate results
        cycle
      else
        broadcast a termination message to MM_{i,j}s
        terminate workers
        store the merged results
        exit
      end if
    end if
  else
    MM_{i,j} receives a message from MM_{i,1}
    if (not a termination message) then
      run one DIRECT iteration and reduce intermediate results
    else
```

exit
end if
end if
end do
$MM_{i,1}$ sends the final results to SM_i

The control mechanism is a two-level messaging—between SM_i and $MM_{i,1}$, and between $MM_{i,1}$ and each $MM_{i,j}$. The DIRECT parameters are passed from SM_i to $MM_{i,1}$, which broadcasts them again to each $MM_{i,j}$. To reduce the control overhead, no handshakes are involved between SM_i and $MM_{i,1}$ before the local stopping criteria are met. However, it is inefficient to dedicate a *SM* process to monitoring the search status and wrapping up the search at the end, causing significant communication overhead.

3.1.2 Task allocation

3.1.2.1 The SELECTION implementation At Level 2, *MM* processes cooperate on identifying convex hull boxes stored in a shared data structure of a global addressing space using GPSHMEM [21]. Two sets of global shared data structures `OPTSET` and `OPTSET_INDEX` (Fig. 5) are used.

The structures `OPTSET` and `OPTSET_INDEX` are allocated and distributed across all *MM* processes, which use one-sided communication operations such as "put" and "get" to access shared data. These one-sided operations provide direct access to remote memory with less interaction between communicating parties. In addition, the data structure `LOCALSET` is allocated at $MM_{i,1}$ for merging the boxes with the same size. When only one *MM* process exists, SELECTION is the same as that in the sequential DIRECT. The following steps describe the SELECTION step implemented in pDIRECT_I.

1. $MM_{i,j}$ $(j = 1, \ldots, n)$ puts all the lowest valued boxes for different box sizes to its own portion in `OPTSET` and updates its index in `OPTSET_INDEX`.
2. $MM_{i,1}$ gets all boxes in `OPTSET` and merges the boxes with the same size to `LOCALSET`.
3. $MM_{i,1}$ finds convex hull boxes in `LOCALSET` and puts a balanced number of boxes for each $MM_{i,j}$ into `OPTSET` (the load balancing algorithm at $MM_{i,1}$ is described in Pseudocode 2).

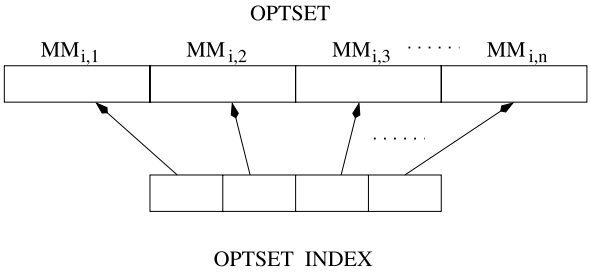


Fig. 5 Data structures in GPSHMEM

4. $MM_{i,j}$ gets its portion of the convex hull boxes from OPTSET, removes some boxes (if any) previously assigned to itself that are now assigned to other $MM_{i,j}$, and starts processing its convex hull boxes.

Each box is tagged with a processor ID and other indexing information to be tracked by its original owner. To minimize the number of local box operations (i.e., removals and additions) and maximize data locality, $MM_{i,1}$ restores the boxes back to their contributors before it starts load adjustment. The shared memory approach can access more memory on multiple machines than on a single machine. However, firstly, it depends on multiple software packages for global addressing, such as GPSHMEM and ARMCI [18]. Secondly, resizing the global data structures to hold an unpredictable number of lowest boxes involves expensive global operations across multiple machines. Thirdly, collecting all lowest boxes at $MM_{i,1}$ burdens local buffer storage as well as network traffic. Lastly, a global barrier is needed between steps to avoid premature "get" operations.

Pseudocode 2

```
N_box: the number of global convex hull boxes
n: the number of MM processes
avgload: the average workload measured in boxes
underload: the workload shortfall from the desired avgload
overload: the workload extra over the desired avgload

merge all boxes from OPTSET to LOCALSET by the box sizes
find convex hull boxes in LOCALSET and update N_box
restore boxes given by MM_{i,j} to its portion in OPTSET
avgload := ⌈(N_box/n)⌉
d := 1 (loop counter)
OUTLOOP: do
  if (d = n) exit OUTLOOP
  if (OPTSET_INDEX(d) < avgload) then
    d_1 := d
    INLOOP: do
      underload := avgload − OPTSET_INDEX(d)
      d_1 := (d_1) mod n
      if (d_1 = d) exit INLOOP
      if (OPTSET_INDEX(d_1) > avgload) then
        overload := OPTSET_INDEX(d_1) − avgload
        if (overload ≥ underload) then
          shift enough load over
          OPTSET_INDEX(d) := avgload
          OPTSET_INDEX(d_1) := OPTSET_INDEX(d_1) − underload
          exit INLOOP
        else
          shift some and look for more
```

```
                OPTSET_INDEX(d) := OPTSET_INDEX(d) + overload
                OPTSET_INDEX(d_1) := avgload
              end if
            end if
          end do INLOOP
        end if
        d := d + 1
      end do OUTLOOP
```

3.1.2.2 Worker assignment As shown in Pseudocode 2, the load adjustment is done at $MM_{i,1}$, which distributes the work to $MM_{i,j}$ by using the shared data structures in GPSHMEM. Then, each *MM* process subdivides its share of convex hull boxes and distributes the function evaluation tasks down to its workers. Although the control mechanism is simple, this centralized strategy suffers a common bottleneck problem. Furthermore, workers are not shared by *MM* processes. A worker is exclusively under the command of a particular *MM* that spawns it at the beginning. This fixed assignment degrades the processor utilization and load balancing among workers.

3.2 pDIRECT_II

The three-level hierarchy in pDIRECT_I is reshaped to be a group-pool structure with subdomain groups of masters and a globally shared pool of workers as shown in Fig. 6. The *SM* and *MM* processes in pDIRECT_I are now grouped together to maintain data structures and perform SELECTION and DIVISION, while globally shared workers perform SAMPLING. This scheme is implemented with a pure message passing model, which removes the dependency on multiple software packages, simplifies the program structure, and improves the parallel performance.

3.2.1 Topology

Each subdomain is served by a subdomain group of masters in lieu of the subdomain master ring. Let $SM_{i,j}$ stand for the master j in subdomain i. $SM_{i,1}$ is the root master for subdomain i. In addition to carrying out common tasks like other masters, the root masters $SM_{i,1}$ $(i = 2, \ldots, m)$ also communicate with $SM_{1,1}$ to finalize the search. This star shaped connection centered at $SM_{1,1}$ has replaced the ring topology in pDIRECT_I.

Moreover, all $SM_{i,j}$ $(j = 1, \ldots, n)$ processes except $SM_{1,1}$ become workers when they have finished all search activities for their subdomain. This dynamic feature reduces the processor idleness and offers an exact stopping condition unavailable in pDIRECT_I. When the stopping condition is satisfied, all workers will receive notifications that all masters have become inactive, so the workers, except those who were converted from masters, will terminate themselves without further messaging. The masters, including those who became workers, will wait for a termination message sent from the root master (processor 0) down along a logical tree of master processors in $\log_2(m \times n)$ steps, where $m \times n$ is the total number of masters. Recall that

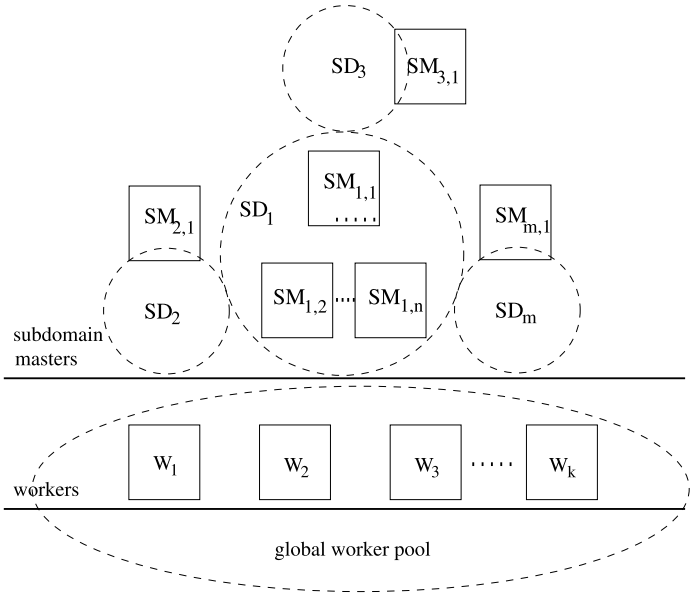


Fig. 6 The parallel scheme for pDIRECT_II

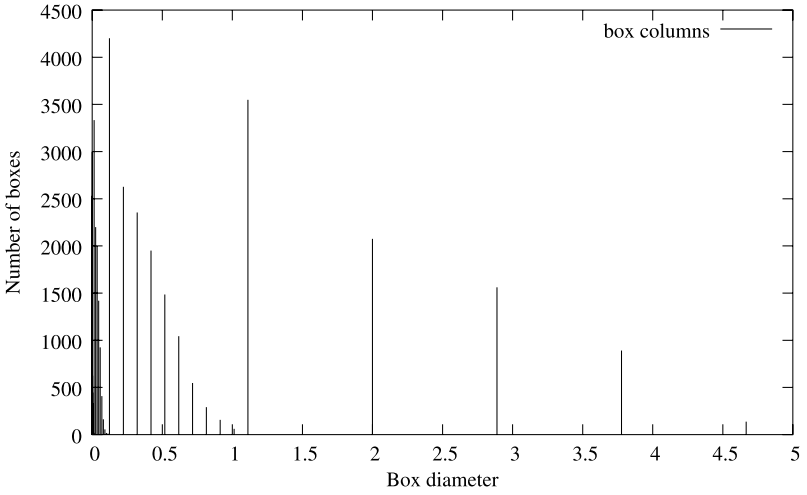
the termination message is passed linearly along the ring and logarithmically down to the $\lfloor M \rfloor$-ary trees in pDIRECT_I. Clearly, the new termination scheme here does not require such a complicated control mechanism as in pDIRECT_I.

3.2.2 Data structures

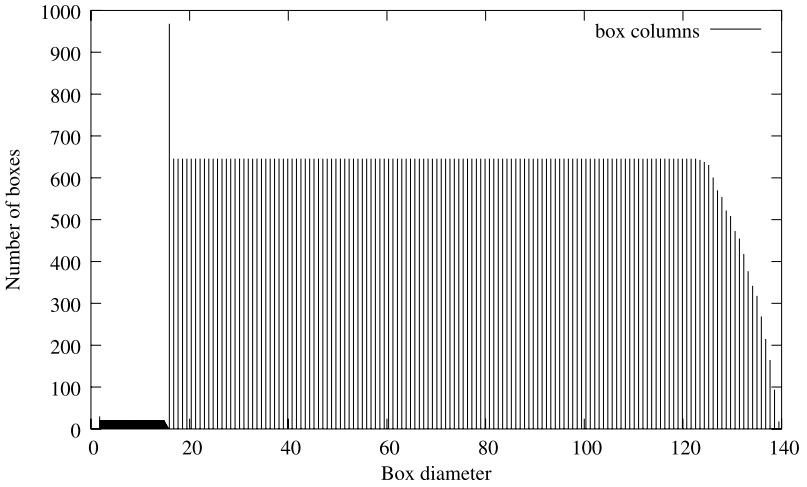
The data structure design directly affects the efficiency of local memory operations as well as global data distribution. The set of dynamic data structures borrowed from [12] was kept the same in pDIRECT_I, but improved in pDIRECT_II. Limiting box columns (LBC) is a technique developed to reduce the memory requirement.

Let $I_{\max}$ be the maximum number of iterations allowed (a stopping criterion), I_{current} be the current iteration number, and C be one of the box columns. Each of the iterations $I_{\text{current}}, \ldots, I_{\max}$ can subdivide at most one box from C, because at most one box from C can be in the set of convex hull boxes at any iteration. Therefore, C only needs to contain at most $L = I_{\max} - I_{\text{current}} + 1$ boxes with the smallest function values. Boxes with larger function values are not considered by the DIRECT search limited to $I_{\max}$ iterations. However, the number of boxes generated per box column is usually much larger than L. Figure 7 shows the box column lengths for (1) Test Function 6 with dimension $N = 10$ and $I_{\max} = 400$ and (2) the budding yeast problem with $I_{\max} = 40$. (These two functions are defined later in Sect. 4.) Most of the box columns are longer than $I_{\max} >= L$ in both (1) and (2). When the stopping criterion $I_{\max}$ is given, storing only L boxes in box columns would significantly reduce the memory demands.

Since each box column is implemented as a min-heap ordered by the function values at the box centers, all box column operations without LBC take $\mathcal{O}(\log n)$ time



(1) Test Function 6 with $N = 10$ and $I_{\max} = 400$.



(2) Budding yeast problem, $N = 143$ and $I_{\max} = 40$.

Fig. 7 Box column lengths at the last iteration $I_{\max}$

and only two types of heap operations are involved—removing boxes with the smallest function values and adding new boxes. Additionally, LBC needs to remove the boxes with the largest function values ($b_{\max}$s). The min-heap data structure requires a $\mathcal{O}(n)$ time algorithm to locate the $b_{\max}$ boxes. In future work, a min-max heap [1] will replace the min-heap data structure to find the $b_{\max}$ boxes with constant time, which makes all operations $\mathcal{O}(\log n)$ time. The min-max heap makes a huge difference when $I_{\max}$ is large, since the number of boxes in a box column heap is very large. The experimental results in Sect. 4 show the improvement of pDIRECT_II over pDIRECT_I in reducing local buffer size.

3.3 Task allocation

Task allocation policies have strong connections to important performance metrics such as parallel efficiency and load balancing. Here, several improvements were made in allocating both box subdivision tasks in the SELECTION step and function evaluation tasks in the SAMPLING step.

3.4 The SELECTION implementation

In subdomain i, SELECTION is accomplished jointly by masters $SM_{i,j}$, $j = 1, \ldots, n$, where n is the total number of subdomain masters per subdomain. When $n = 1$, SELECTION is the same as that in the sequential DIRECT. When $n > 1$, SELECTION is done in parallel over the index i as follows.

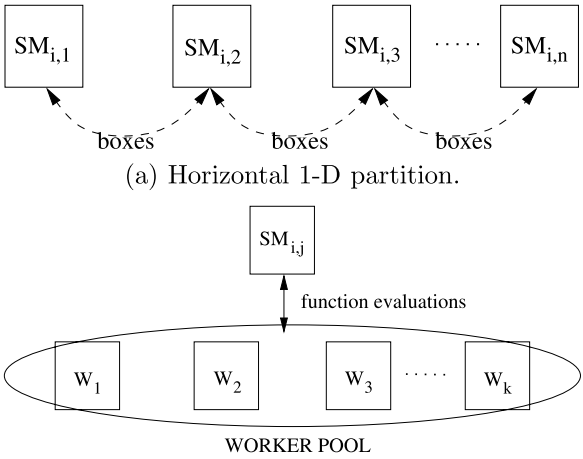
1. The $SM_{i,j}$ identify local convex hull box sets $S_{i,j}$, $j = 1, \ldots, n$.
2. $SM_{i,1}$ gathers the $S_{i,j}$ from all the $SM_{i,j}$.
3. $SM_{i,1}$ merges the $S_{i,j}$ by box diameters and finds the global convex hull box set S_i.
4. All the $SM_{i,j}$ receive the global set S_i and find their portion of the convex hull boxes.

All buffers used here are locally allocated and resized incrementally according to the updated number of boxes involved in the convex hull computation. Note that all boxes in the global convex hull box set S_i must also be in the union of the sets of local convex hull boxes ($S_{i,j}$) of the masters. Therefore, each master $SM_{i,j}$ computes $S_{i,j}$ in parallel and $SM_{i,1}$ only considers the union of all $S_{i,j}$ instead of all the lowest boxes with different diameters, as was done in pDIRECT_I. This decentralized SELECTION implementation reduces memory requirements for buffers, as well as the amount of data transferred over the network. However, a large number of subdomain masters will not perform well due to the global communication and synchronization required for finding convex hull boxes at every iteration. Reference [23] describes a sampling technique that can further reduce the bandwidth requirements, but it comes at the expense of requiring another global communication round. Another possibility would be for $SM_{i,1}$ to gather via a d-way tree only the final merged $S_{i,j}$, where each intermediate tree node does a partial convex hull merge of its d (merged) inputs (the optimal value for d is derived in [23]).

3.4.1 Task partition

pDIRECT_II supports two ways of task partitioning. In the case of low computation cost, communication overhead dominates the parallel execution time and overshadows the benefits in distributing function evaluation tasks to workers. Wisely, masters would rather keep computation locally and simply share the memory burden with other masters without any worker involved in the picture. This is called horizontal 1-D partition for box subdivision tasks (Fig. 8a). Experimental results in Sect. 4 show that this scheme achieves better speedup (for cheap function evaluations) than the vertical 1-D partition for function evaluation tasks (shown in Fig. 8b). When these two ways are combined, they become a hybrid 2-D partition. This hybrid scheme is

Fig. 8 Task partition schemes



(a) Horizontal 1-D partition.

(b) Vertical 1-D partition.

usually preferred for the following reasons. First, the computation cost is higher or at least comparable to the communication cost in most real world design problems. Generally, overlapping the computation on worker processors in the vertical 1-D partition is a reasonable approach. Second, the data sharing scheme in the horizontal 1-D partition relieves the heavy memory burden on a single master processor for solving a large scale and/or high dimensional problem.

3.4.2 Worker assignment

The worker pool is globally shared by all masters in all subdomain groups. Each worker polls all (selected) masters for evaluation tasks and returns the function values. Workers proceed in cycles that roughly correspond to the DIRECT iterations. During each cycle, a worker requests tasks from a randomly selected subset of masters until all of them are out of work. This is called the "nonblocking" loop in Pseudocode 3. Once the cycle is over, the worker blocks waiting (the "blocking" loop) for further instructions from a fixed master, which is selected such that every master has a fair number of blocked workers waiting in the queue. As each worker loops through receiving, sending, and processing messages of various types, think of the worker as being in one of two states, described by the values "blocking" or "nonblocking". These states correspond to whether or not the worker is blocked from receiving work requests. Pseudocode 3 below describes how a worker W_i evaluates the objective function for masters $SM_{i,j}$ ($i = 1, \ldots, m$ and $j = 1, \ldots, n$) during SAMPLING.

Pseudocode 3

m: the number of subdomains, given as input
n: the number of masters per subdomain, given as input
converted: `TRUE` when W_i was a master
loop: the loop status

```
C_active: the counter for the total number of "active" masters
    that have not reached the last iteration
C_idle: the counter for idle masters
P_wi: the processor ID (PID) of W_i
mesg: the message received from others
loop := "nonblocking"
C_active := current total number of active masters
C_idle := 0 (assume all masters busy initially)
OUTLOOP: do
   if (loop = "nonblocking") then
      send a nonblocking request to a randomly selected master SM_{i,j}
         from C_active − C_idle busy masters
   else
      set all masters to status busy
      C_idle := 0
      send a blocking request to the master SM_{i,j} that ranks as
         (P_wi − C_active) mod C_active in the list of all active masters
   end if
   INLOOP: do
      keep waiting for any messages
      select case (mesg)
         case (an evaluation task from SM_{i,j})
            evaluate all points in the task
            send the results back to SM_{i,j}
            if (SM_{i,j} is responding to a blocking request) then
               loop := "nonblocking"
            end if
         case ("no point" from SM_{i,j})
            if (SM_{i,j} is at status busy) then
               set SM_{i,j}'s status idle
               C_idle := C_idle + 1
            end if
            exit INLOOP
         case ("all done" from SM_{i,j})
            if (SM_{i,j} is at status busy) then
               set SM_{i,j}'s status idle
               C_idle := C_idle + 1
            end if
            remove SM_{i,j} from the master list
            C_active := C_active − 1
            if (C_active = 0) then
               if (converted = TRUE) then
                  cycle INLOOP (will wait for "terminate")
               else
                  exit OUTLOOP (terminate itself)
               end if
```

```
            end if
            exit INLOOP
        case (a "non-blocking" request from a worker)
            reply "all done" to this worker (W_i was a master)
        case (a "blocking" request from a worker)
            reply "all done" to this worker (W_i was a master)
        case ("terminate" from the parent processor)
            pass "terminate" to the left and right children (if any)
            exit OUTLOOP
    end select case
  end do INLOOP
  if (C_active = C_idle) then
    loop := "blocking"
  else
    loop := "nonblocking"
  end if
end do OUTLOOP
```

At the same time, the master $SM_{i,j}$ is generating sample points and responding to worker requests as described in the following Pseudocode 4.

Pseudocode 4

```
S_{i,j}: the portion of global convex hull boxes for SM_{i,j}
Q_w: the blocked worker queue
A_b(1 : k) := 0 (the array of counters for tracking the number of blocking
  requests from workers W_1, W_2, ..., W_k)
C_new := 0 (the counter for new points)
C_send := 0 (the counter for points that have been sent to workers)
C_eval := 0 (the counter for evaluated new points)
N_bin: the upper limit on the number of evaluation tasks sent to a worker
  at one time
mesg: the message received from others
if (S_{i,j} is empty) then
  release all blocked workers in Q_w (send them "no point" messages)
else
  find the longest dimensions for all boxes in S_{i,j}
  C_new := the number of all newly sampled points along longest dimensions
  if (Q_w is not empty) then
    loop sending at most N_bin number of points to each worker in Q_w
      with a tag "responding to your blocking request"
    update C_send
    release the remaining blocked workers (if any)
  end if
  do while (C_eval < C_new)
    keep waiting for any messages
```

```
select case (mesg)
  case (a "non-blocking" request from W_i)
    if (C_send < C_new) then
      send at most N_bin number of points to W_i
      update C_send
    else
      send "no point" message to W_i.
    end if
  case (a "blocking" request from W_i)
    if (C_send < C_new) then
      send at most N_bin number of points to W_i with a tag
                    "responding to your blocking request"
      update C_send
    else
      if (A_b(i) = 0) then
        A_b(i) := 1
        send "no point" to W_i
      else
        put W_i into Q_w
      end if
    end if
  case (function values from W_i)
    save the values and update C_eval
    if (C_send < C_new) then
      send W_i another task with at most N_bin points
      update C_send
    else
      send "no point" to W_i
    end if
end select case
end do
end if
```

At the beginning of each iteration, $SM_{i,j}$ sends evaluation tasks to its blocked workers. If it has more blocked workers than tasks, it signals the remaining blocked workers to start a new cycle of requesting work from other masters. Otherwise, $SM_{i,j}$ keeps receiving function values from workers and sending out more tasks. When $SM_{i,j}$ is out of tasks, it notifies workers that are requesting tasks and queues up the workers that are blocked waiting for the next iteration. An array of blocking status (A_b) is used to track the number of times that a worker has sent a blocking request to this master during this iteration. After the first blocking request from a worker, $SM_{i,j}$ tells the worker to continue seeking work from other masters. After the second blocking request from that same worker, during this iteration, $SM_{i,j}$ queues up that worker; this gives workers a better chance to find masters who have just become busy. Observe that the subdomain masters within the same subdomain need to synchronize with each other to find global convex hull boxes during every iteration; however,

no synchronization or communication is needed among workers, and masters from different subdomains also work independently, until the final termination starts.

When all the masters are out of work at the end of an iteration, the next iteration begins, and the masters from different subdomains may start the next iteration at different times. Therefore, a master should encourage a worker, who has sent it the first blocking request, to seek work again from other masters. This asynchronous design allows a large number of workers to be used efficiently across masters and subdomains. Empirical results have shown that workers achieve a better load balance for a multiple subdomain search than for a single domain search. In comparison, workers in pDIRECT_I work only for a fixed master, so they have to sit idle when the master runs out of work until the next iteration starts.

This scheme also naturally distributes tasks to workers according to the speed at which they finish the work, unlike the load balancing methods that attempt to distribute an approximately equal number of function evaluation tasks to each worker. These methods assume that (1) the function evaluation at different coordinates costs the same computationally and/or (2) each worker finishes the function evaluation within the same amount of time. In fact, these two assumptions are not satisfied in many parallel systems, even though some are claimed to be homogeneous. Most importantly, many engineering design problems do have different computation cost for different regions. Therefore, the measure of a reasonable load balancing should not be the equal quantities of tasks that are distributed among workers, but the degree that all of the workers are kept busy. Note that this scheme adds a parameter N_{bin} used for stacking function evaluations to one evaluation task. It reduces the communication overhead when the objective function is cheap. However, N_{bin} should be set to 1 if the objective function is expensive. Otherwise, fewer tasks are available to workers and a load imbalance occurs.

4 Performance results

This section presents performance results regarding the main design issues discussed in the last section. The test functions used are described in Table 1; the first seven have the same initial domain $[-2, 3]^N$. For some experiments, dimension $N = 150$ and an

Table 1 Test functions

#	Description
1	$f = x \cdot x/3000$
2	$f = -\sqrt{\sum_{i=1}^{N} x_i - 0.5(i-1)/N}$
3	$f = 1 + \sum_{i=1}^{N} {x_i}^2/500 - \prod_{i=1}^{N} \cos(x_i/\sqrt{i})$
4	$f = \sum_{i=1}^{N} 2.2 \times (x_i + 0.3)^2 - (x_i - 0.3)^4$
5	$f = \sum_{i=1}^{N} \sum_{j=1}^{i} {x_j}^2$
6	$f = \sum_{i=1}^{N} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$
7	$f = 10N + \sum_{i=1}^{N} x_i^2 - 10\cos(2\pi x_i)$
8	Budding yeast parameter estimator [20]

Table 2 Comparison of experiments without (NON-LBC) and with LBC for Test Functions 1–7

#	NON-LBC		LBC		
	I_{out}	$C_{\text{real}}/10^6$	$C_{\text{real}}/10^6$	% diff.	I_{out}
1	159	153	112	26	273
2	79	164	77	52	647
3	213	162	111	31	391
4	90	163	82	50	1000
5	286	164	124	24	467
6	163	160	109	31	328
7	78	162	85	47	377

artificial time delay T_f are used to make the first seven test functions comparable to the 143-parameter estimation problem from computational biology. Large scale performance results for Problem 8 and their biological significance are given in [20].

4.1 Data structures

The size of the data structures is the number C_{real} of 64-bit REAL variables in the box data structures. Table 2 compares I_{out} (the number of iterations when the memory is used up) with and without LBC and computes the percentage of C_{real} reduction in LBC for runs with $I_{\max} = I_{\text{out}}$ without LBC. All tests failed to allocate memory when C_{real} reaches $\approx 150 \times 10^6$ on System X. LBC reduces the size of the data structures by 20–50% for all the test functions 1–7 and by 37% for the budding yeast problem. As the number of box columns grows larger without limit, a single master runs fewer iterations without LBC than with LBC.

4.2 Task allocation

The following experiments demonstrate the effectiveness of the SELECTION implementation, task partition, and worker assignment in pDIRECT_II.

4.2.1 SELECTION efficiency

Table 3 compares N_{gc} (number of global convex hull boxes), N_{lc} (the combined number of local convex hull boxes), and N_d (the combined number of different box diameters) on 32 subdomain masters for all test functions at the last iteration ($I_{\max} = 1000$ for Test Function 1–7 and $I_{\max} = 100$ for the budding yeast problem). In pDIRECT_I, $MM_{i,1}$ collects N_d boxes and finds the convex hull box set. In pDIRECT_II, N_{lc} local convex hull boxes are found by $SM_{i,j}$ ($j = 1, \ldots, n$), then gathered on $SM_{i,1}$, which identifies the global convex hull boxes. This approach increases the concurrency of the SELECTION implementation. Table 3 shows that it reduces the amount of data transferred over the network and the buffer size by 50–90% for all test functions.

Table 3 Comparison of N_{gc}, N_{lc}, and N_d at the last iteration I_{max}

#	N_{gc}	N_{lc}	N_d	$\frac{(N_d - N_{lc})}{N_d}$
1	58	2605	33358	92%
2	89	1228	6805	81%
3	145	2056	22375	90%
4	3	3201	6756	52%
5	140	1830	20276	90%
6	144	1276	28003	95%
7	144	3531	20614	82%
8	20	159	6545	97%

4.3 Task partition

The following experiment is to study the parallel performance of the horizontal and vertical 1-D partition schemes. A total of $P = 288$ processors are used. In the horizontal partition, each run has all P masters that evaluate objective functions locally. In the vertical partition, a single master sends evaluation tasks to $P - 1$ workers, each task holding $N_{bin} = 1$ set of point coordinates. Table 4 shows the timing results for Test Function 6 with $N = 150$, given two function evaluation costs: $T_f = 0.1$ second (the artificial case) and the original cost $T_f = 0.0$ (less than 1.0E-7 second). Set $I_{max} = 300$ for the original cost and $I_{max} = 90$ for the artificial cost. To ensure that the two partition schemes are comparable in the number of processors, the timing results are measured from $P = 3$ processors (if $P = 1$, no workers are used in the vertical 1-D partition). Note that the horizontal 1-D partition has all P processors available for function evaluations, while the vertical 1-D partition has only $P - 1$ processors for that. Therefore, the parallel efficiency is estimated with the base $= 3$ processors for the horizontal 1-D partition and base $= 2$ for the vertical 1-D partition. The efficiency E is thus computed as $(T_{base}/T_P)/(P/\text{base})$ as shown in Fig. 9.

When $T_f = 0.0$ second, the horizontal 1-D partition (HP_0) runs faster than the vertical 1-D partition (VP_0). The masters in HP_0 locally evaluate the cheap objective function, thus avoiding the communication overhead for talking to workers. However, HP_0 does not achieve any speedup for $P = 72$, 144, and 288, because the number of convex hull boxes is insufficient to keep all masters busy (some masters have no convex hull boxes, thus evaluating no points locally). As for VP_0, the communication overhead always dominates the execution cost and it fails to achieve any speedup after P reaches 144. When $T_f = 0.1$ second, the vertical 1-D partition (VP_1) runs more efficiently than the horizontal 1-D partition (HP_1) (see Fig. 9) except for $P = 3$. When $P = 3$, three masters are evaluating functions for HP_1, while only two workers are doing so for VP_1. Nevertheless, all runs with $P \geq 5$ of VP_1 take a much shorter time than those of HP_1. The first reason is that the communication overhead is negligible compared to the objective function cost for VP_1. Secondly, when P is large, no convex hull boxes are assigned to some masters, so they have to sit idle in the case of HP_1. In general, the number of convex hull boxes is much smaller than the number of function evaluations, because DIRECT samples two new points along each longest dimension for every convex hull box. Hence, the

Table 4 Parallel timing results (in seconds) for different function evaluation costs T_f for Test Function 6 with $N = 150$

	Horizontal 1-D partition							
$T_f \backslash P$	3	5	9	18	36	72	144	288
0.0	21.04	15.36	12.03	12.78	14.88	40.71	23.37	31.49
0.1	6785.88	4614.23	2741.11	2633.46	2634.53	2636.03	2652.78	2644.96
	Vertical 1-D partition							
$T_f \backslash P$	3	5	9	18	36	72	144	288
0.0	49.93	41.18	40.28	41.81	45.10	45.53	65.10	55.07
0.1	8720.27	4361.30	2184.76	1033.28	507.14	256.09	149.95	79.27

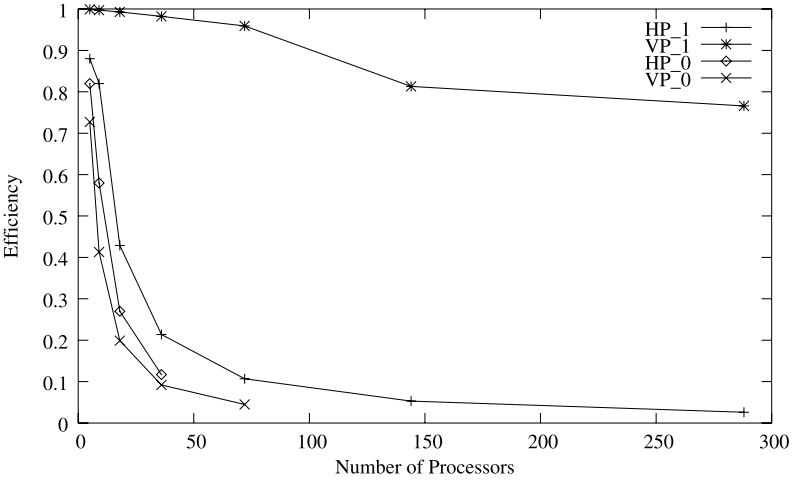


Fig. 9 Comparison of the parallel efficiencies with different partition schemes and objective function costs

pure horizontal 1-D partition reaches its limits when P is greater than the number of convex hull boxes. On the other hand, the memory limit on a single master makes the pure vertical 1-D partition impossible for runs with large $I_{\max}$. Therefore, a hybrid partition scheme is generally preferable to the pure partition schemes.

In the following two experiments, several hybrid partition schemes with different numbers of masters and workers are compared with the single master scheme for Test Function 6 with $N = 150$ and $T_f = 0.1$ second for a single subdomain. The first experiment varies the number of masters (2^i, $i = 0, \ldots, 5$) and fixes the total number of processors (implicitly, the number of workers also changes). The second experiment varies the number of masters and fixes the number of workers. $P = 100$ (total number of processors) is used for Experiment 1. 100 and 200 workers, respectively, are involved in Experiment 2. The measured parallel execution timing results listed

Table 5 Comparison of theoretical parallel execution time T_t and the parallel timing results T_p with different hybrid partition schemes for Test Function 6 with $N = 150$, $T_f = 0.1$ sec, and $I_{\max} = 90$. P_{100} stands for using a total of 100 processors. W_{100}, W_{200} stand for using a total of 100, 200 workers, respectively. The overhead is $T_o = T_p - T_t$

Schemes		Number of Masters					
		1	2	4	8	16	32
P_{100}	T_p	203.42	204.20	207.79	215.44	234.15	282.38
	T_t	180.10	181.70	185.80	192.90	211.40	260.20
	T_o	23.32	24.10	26.09	29.64	41.25	22.18
	E_f	88.5%	88.9%	89.4%	89.5%	90.3%	92.1%
W_{100}	T_p	201.53	185.41	184.83	184.88	185.61	187.29
	T_t	178.00	178.00	178.00	178.00	178.00	178.00
	T_o	23.53	7.41	6.83	6.88	7.61	9.29
	E_f	88.3%	96.0%	96.3%	96.3%	95.9%	95.0%
W_{200}	T_p	102.32	102.06	101.56	101.55	103.14	105.86
	T_t	91.30	91.30	91.30	91.30	91.30	91.30
	T_o	11.02	10.76	10.26	10.25	11.84	14.56
	E_f	89.2%	89.5%	89.9%	89.9%	88.5%	86.2%

in Table 5 are compared to a theoretical lower bound defined by

$$T_t = \sum_{i=1}^{I_{\max}} \left\lceil \frac{N_i}{k} \right\rceil T_f,$$

where T_t is the theoretical lower bound on the parallel execution time, N_i is the number of function evaluation tasks at iteration i, and k is the total number of workers. T_t depends on both the problem and the computing resource. It assumes all workers are fed continuously with evaluation tasks, distinct from reality, where finding convex hull boxes, synchronization, and communication all cost time as well. Clearly when N_i is not exactly a multiple of k, some workers are idle during the last working cycle for that iteration. A worker ideally obtains either $\delta_+ = \lceil N_i/k \rceil$ or $\delta_- = \lfloor N_i/k \rfloor$ number of tasks. The number X of idle workers can be derived from

$$\delta_+(k - X) + \delta_-(X) = N_i.$$

Here, define the overhead of function evaluation $T_o = T_p - T_t$ and the efficiency of function evaluation $E_f = T_t/T_p$, where T_p is the measured parallel execution time.

Figure 10 shows how the number of new evaluations per iteration (N_i) changes for Test Function 6 with $N = 150$ and $I_{\max} = 90$. Given the number of workers, T_t is then computed in Table 5, which shows that a wide range of hybrid task partition schemes perform reasonably well ($86.2\% \le E_f \le 96.3\%$). In the case of P_{100}, E_f grows slightly as the number of masters increases up to 32. Clearly, the number of convex hull boxes is sufficient to keep 32 masters busy. Moreover, smaller master-to-worker ratios seem to correspond to lower E_f values. This phenomenon may indicate

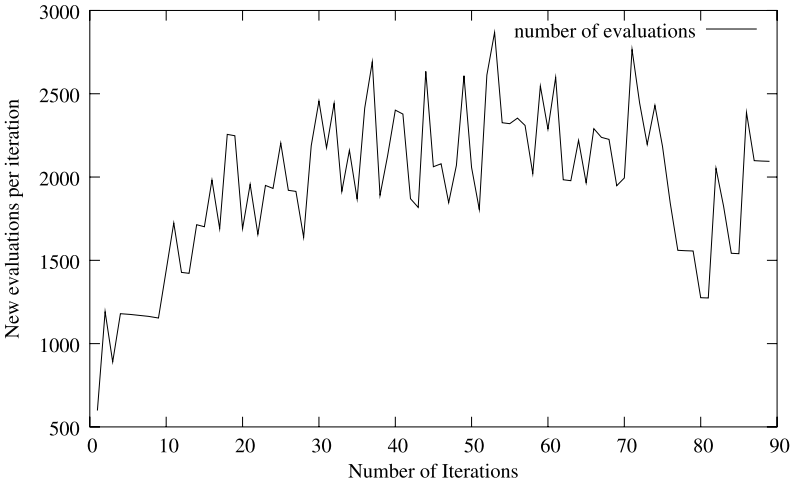


Fig. 10 The plot of N_i $(i = 1, \ldots, 90)$ for Test Function 6 with $N = 150$

a potential bottleneck problem at masters that communicate with a great number of workers, or simply more idle workers. The above supposition was further investigated in the second experiment. In both W_{100} and W_{200}, E_f is improved at the beginning as the number of masters increases to a "peak point" with the best E_f. Then, it is degraded when the synchronization overhead among masters starts to dominate. Note that the peak point is 4 for W_{100} and 8 for W_{200}, while the master-to-worker ratios at these two points are the same, 1:25. Moreover, W_{200} has lower E_f values than W_{100} for the same number of masters, because masters in W_{200} deal with more workers, and more workers in W_{200} are likely to be idle. Also, the same amount of work distributed to 100 workers in W_{100} generates more communication interactions between masters and workers in W_{200}. Therefore, the search with a single subdomain will always eventually decrease E_f as more and more workers are used.

4.3.1 Worker assignment

The following experiment demonstrates that function evaluation tasks are allocated more efficiently for a multiple subdomain search than for a single subdomain search. Load balancing among workers is improved greatly with the globally shared worker pool of pDIRECT_II, especially when masters of different subdomains generate unequal amounts of work. $P = 320$ processors were used for the budding yeast problem. $P = 200$ processors were used for the artificial test problems (Test Functions 1–7) with $N = 150$ and $T_f = 0.1$ sec.

In this experiment, the feasible domain is split into four subdomains. Note that the same test problem with four split subdomains can be solved in three different ways using P processors.

(a) All P processors are used to run a multiple subdomain search with four subdomains. The parallel execution time is T_a.

(b) Four single subdomain searches are run in parallel, each using $P/4$ processors on one of the four subdomains. The overall parallel execution time T_b is the longest duration of all four runs.
(c) Four single subdomain searches are run sequentially, each using all P processors on each of the four subdomains. The parallel execution time T_c is the total duration of these four runs.

Table 6 compares T_a, T_b, and T_c for all the test problems ($I_{\max} = 90$ for Test Functions 1–7 and $I_{\max} = 40$ for the budding yeast problem). T_a is the smallest among the three. T_b is only slightly bigger than T_a for Test Functions 2, 3, 4, 5, and 7, but becomes significantly bigger (>5%) for Test Functions 1, 6, and 8, each of which has a large s^2, the variance of the total number of function evaluations for the four subdomains (in Table 7). Also, T_c is the largest among the three. Observe that it is only slightly bigger (< 5%) than T_a for Test Functions 2 and 7. Table 7 shows that these two test functions have a large $\bar{e} = \sum e_i / I_{\max}$ (>6000), the average number of function evaluations per iteration, where e_i is the total number of evaluations for subdomain i. Since more tasks are generated at each iteration for Test Functions 2 and 7 than for the other test functions, $P - 1$ workers are better load balanced in case (c).

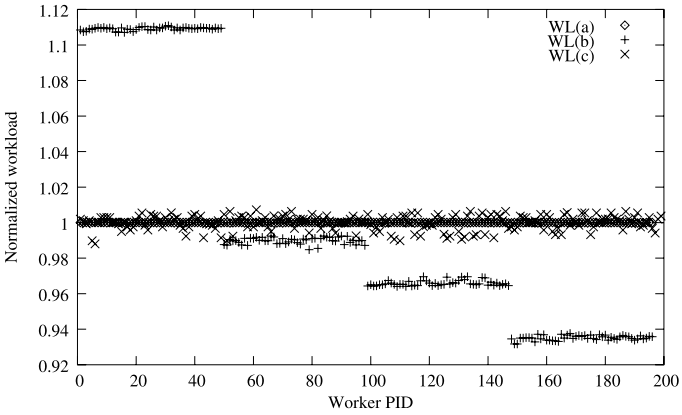
Figure 11 shows the normalized workload among workers for two runs: (1) Test Function 6 with moderate $\bar{e}$ and large s^2, and (2) Test Function 7 with large $\bar{e}$ and small s^2. Generally, workload is normalized by dividing the total evaluation time for

Table 6 Comparison of T_a, T_b, and T_c in seconds

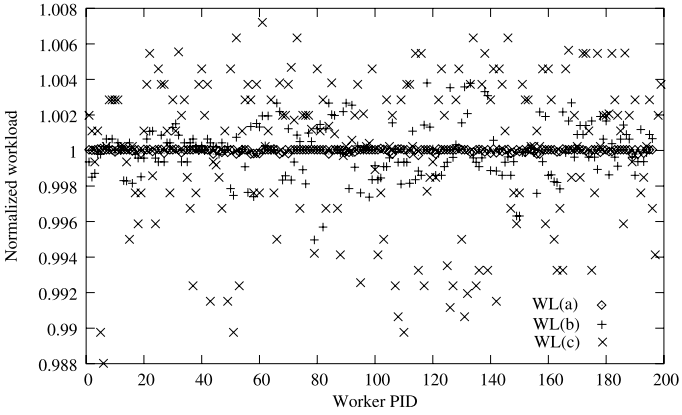
#	T_a	T_b	T_c	$\frac{(T_b - T_a)}{T_a}$	$\frac{(T_c - T_a)}{T_a}$
1	382	407	441	6.2%	15.4%
2	1132	1139	1185	0.6%	4.6%
3	358	369	417	3.1%	16.4%
4	870	874	921	0.4%	5.9%
5	260	263	312	1.1%	20.0%
6	428	476	477	11.2%	11.4%
7	1142	1148	1196	0.5%	4.7%
8	8595	10643	11059	23.8%	28.7%

Table 7 Comparison of total number of subdomain function evaluations for experiments listed in Table 6. e_i is the total number of evaluations for subdomain i, which is the same for all of the ways (a), (b), and (c)

#	e_1	e_2	e_3	e_4	$\bar{e}$	s^2
1	181409	194927	194927	181463	2090.9	7789.1^2
2	550691	550691	550691	550691	6118.8	0
3	176075	176075	176075	176075	1956.4	0
4	421723	421723	421723	421723	4685.8	0
5	123685	123685	123685	123685	1374.3	0
6	228193	203635	198727	192397	2286.0	15661^2
7	555435	555435	555435	555435	6171.5	0
8	82471	44631	47531	87063	1635.6	22445^2

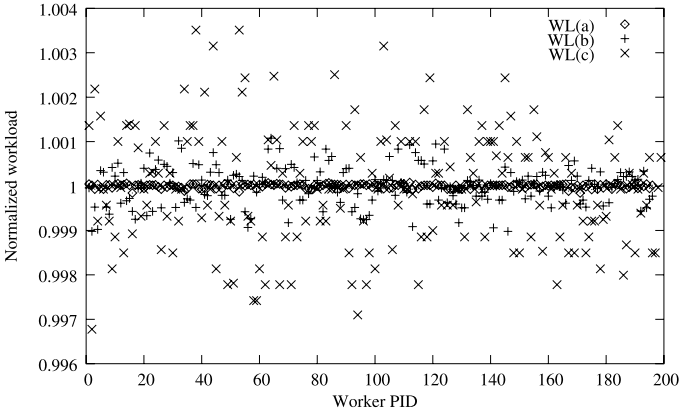


I. Normalization based on P processors for case (b).



II. Normalization based on $P/4$ processors for case (b).

(1) Test Function 6.



(2) Test Function 7.

Fig. 11 Comparison of the workload (WL) patterns among workers for cases **a** (*circles*), **b** ("+" *marks*), and **c** ("x" *marks*)

Table 8 Normalized workload ranges (WR) of a, b, and c for experiments listed in Table 6

#	WR_a	WR_{b1}	WR_{b2}	WR_c
1	[0.99969,1.0002]	[0.95743,1.0413]	[0.99193,1.0091]	[0.97495,1.0250]
Width	5.4685E-5	8.3845E-2	1.7152E-2	5.0023E-2
2	[0.99985, 1.0002]	[0.99871,1.0014]	[0.99871,1.0014]	[0.9959,1.0049]
Width	3.2897E-4	2.6460E-3	2.6460E-3	9.7536E-3
3	[0.99972,1.0001]	[0.99633,1.0038]	[0.99633,1.0038]	[0.98213,1.0138]
Width	3.8E-4	7.4827E-3	7.4827E-3	3.1646E-2
4	[0.99989,1.0001]	[0.99840,1.0017]	[0.99841,1.0017]	[0.99470,1.0032]
Width	2.1E-4	3.2577E-3	3.2576E-3	8.4916E-3
5	[0.99967,1.0005]	[0.99476,1.0062]	[0.99478,1.0063]	[0.98949,1.0104]
Width	7.9529E-4	1.1487E-2	1.1488E-2	2.0914E-2
6	[0.99973,1.0001]	[0.93170,1.1108]	[0.99497,1.0038]	[0.98802,1.0072]
Width	4.0281E-4	1.7910E-1	8.8312E-3	9.9186E-1
7	[0.99984,1.0001]	[0.99898,1.0011]	[0.99898,1.0011]	[0.99678,1.0035]
Width	2.3958E-4	2.0986E-3	2.0986E-3	6.7307E-3

each worker by the average evaluation time for all workers. Specially for case (b), the workload is also computed based on $P/4$ processors for each run instead of considering all P processors for the general normalization. Table 8 lists the normalized workload ranges, where WR_{b1} was obtained by averaging the workload based on P processors and WR_{b2} was obtained by averaging the workload based on $P/4$ processors.

Figure 11 plots the normalized workload among workers for Test Functions 6 and 7 in cases (a), (b), and (c). Figure 11(1)I and II use different ways of normalization for case (b). In all three pictures of Fig. 11, case (a) has the best load balancing, i.e., the workload values fluctuate in the narrowest range around the average value 1.0 as listed in Table 8. In Fig. 11(1)I, case (b) presents the widest range (WR_{b1}) and an interesting pattern that correlates with the variance of the number of function evaluations for the four subdomains of Test Function 6. In case (c), the workload values fall within ranges slightly wider than those in case (a). Nevertheless, if the average workload is computed separately (based on $P/4$ processors) for each run of case (b), case (b)'s *thus computed* average workload range (WR_{b2}) for Test Function 6 is slightly narrower than that of case (c), but wider than that of case (a). This explains the timing results for Test Function 6 ($T_a < T_b \approx T_c$). Since s^2 is 0 for Test Function 7, the workload pattern is regular for all four runs of case (b) as shown in Fig. 11(2). Also, the same workload range is obtained by either averaging the workload of all workers (WR_{b1}) or of the workers within each run (WR_{b2}) of case (b). Similarly for Test Function 7, the narrower workload range implies shorter parallel execution time. This experiment concludes that the multiple subdomain search has the best parallel performance in terms of parallel execution time and load balancing. Multiple subdomain search allows masters from different subdomains to provide work to the globally shared workers at different times, especially for subdomains that generate different amounts of work. In comparison, all masters run out of tasks at

the end of each iteration in the single subdomain search. In the latter case, therefore, all workers will be idle until new tasks are available at the next iteration, a direct consequence of the DIRECT algorithm's data dependency.

5 Conclusions and future work

This paper describes the pertinent considerations and rationale during the evolution of several massively parallel DIRECT implementations. Several memory reduction techniques and scalability improvements in the parallel scheme have been used in the largest application of DIRECT known to the authors—solving 143-dimensional optimization problems on up to 320 processors in parallel. Several design decisions were analyzed and supported by experiments. Future research in parallel DIRECT may consider the following topics. First, memory reduction techniques that take advantage of the limit on the number of iterations can be explored further. In particular, one may consider limits that take several box columns into account. Second, one may try to mitigate the effect of data dependency by guessing which points DIRECT will sample in the next iteration, and proactively sampling them when many workers are idle at the end of the current iteration. Intuitively, several points whose values are yet unknown should not substantially affect the convex hull, so one may be able to precompute certain objective function values ahead of time (effectively for free, by using otherwise idle workers), and replace expensive function evaluations with cheap table lookups later. Third, improving the implementation efficiency of convex hull computation and memory reduction techniques will likely allow parallel DIRECT to scale to even larger problems on even larger machines. This is especially important when the objective function is cheap, and thus the overhead of internal bookkeeping is significant.

Acknowledgement This work was supported in part by AFRL Grant F30602–01–2–0572 and AFOSR Grant F49620-02-1-0090.

References

1. Atkinson, M.D., Sack, J.-R., Santoro, N., Strothotte, T.: Min-max heap and generalized priority queues. Commun. ACM **29**(10), 996–1000 (1986)
2. Baker, C.A., Watson, L.T., Grossman, B., Haftka, R.T., Mason, W.H.: Parallel global aircraft configuration design space exploration. In: Tentner, A. (ed.) High Performance Computing Symposium 2000, Soc. for Computer Simulation Internat, San Diego, CA, pp. 101–106 (2000)
3. Bartholomew-Biggs, M.C., Parkhurst, S.C., Wilson, S.P.: Global optimization approaches to an aircraft routing problem. EUR J. Oper. Res. **146**(2), 417–431 (2003)
4. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann, San Francisco (2001)
5. Cray Research, Inc.: SHMEM Technical Note for C, SG-2516 2.3 (October 1994)
6. Esposito, W.R., Floudas, C.A.: Global optimization in parameter estimation of nonlinear algebraic models via the Error-In-Variables approach. Ind. Eng. Chem. Res. **37**, 1841–1858 (1998)
7. Gablonsky, J.M.: Modifications of the DIRECT algorithm. PhD thesis, Department of Mathematics, North Carolina State University, Raleigh, NC (2001)
8. Gau, C., Stadtherr, M.A.: Nonlinear parameter estimation using interval analysis. In: AIchE Symposium, vol. 94, no. 320, pp. 445–450 (1999)

9. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Pearson Education, Upper Saddle River (2003)
10. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge (1999)
11. He, J., Verstak, A., Watson, L.T., Rappaport, T.S., Anderson, C.R., Ramakrishnan, N., Shaffer, C.A., Tranter, W.H., Bae, K., Jiang, J.: Global optimization of transmitter placement in wireless communication systems. In: Tentner, A. (ed.) Proc. High Performance Computing Symposium 2002, Soc. for Modeling and Simulation International, San Diego, CA, pp. 328–333 (2002)
12. He, J., Watson, L.T., Ramakrishnan, N., Shaffer, C.A., Verstak, A., Jiang, J., Bae, K., Tranter, W.H.: Dynamic data structures for a direct search algorithm. Comput. Optim. Appl. **23**(1), 5–25 (2002)
13. He, J., Sosonkina, M., Shaffer, C.A., Tyson, J.J., Watson, L.T., Zwolak, J.W.: A hierarchical parallel scheme for global parameter estimation in systems biology. In: Proc. 18th Internat. Parallel & Distributed Processing Symp., CD-ROM, IEEE Computer Soc., Los Alamitos, CA (2004)
14. He, J., Sosonkina, M., Watson, L.T., Verstak, A., Zwolak, J.W.: Data-distributed parallelism with dynamic task allocation for a global search algorithm. In: Parashar, M., Watson, L. (eds.) Proc. High Performance Computing Symposium 2005, Soc. for Modeling and Simulation Internat., San Diego, CA, pp. 164–172 (2005)
15. Jones, D.R.: The DIRECT global optimization algorithm. In: Encyclopedia of Optimization, vol. 1, pp. 431–440. Kluwer Academic, Boston (2001)
16. Jones, D.R., Perttunen, C.D., Stuckman, B.E.: Lipschitzian optimization without the Lipschitz constant. J. Optim. Theory Appl. **79**(1), 157–181 (1993)
17. Moles, C.G., Mendes, P., Banga, J.R.: Parameter estimation in biochemical pathways: a comparison of global optimization methods. Genome Res. **13**, 2467–2474 (2003)
18. Nieplocha, J., Carpenter, B.: ARMCI: a portable remote memory copy library for distributed array libraries and compiler run-time systems. In: 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium, IPPS/SDP'99, CDROM (1999)
19. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global arrays: a nonuniform memory access programming model for high-performance computers. J. Supercomput. **10**(2), 169–189 (1996)
20. Panning, T.D., Watson, L.T., Allen, N.A., Chen, K.C., Shaffer, C.A., Tyson, J.J.: Deterministic global parameter estimation for a model of the budding yeast cell cycle, J. Glob. Optim. (to appear)
21. Parzyszek, K., Nieplocha, J., Kendall, R.A.: A generalized portable SHMEM library for high performance computing. In: 12th IASTED International Conference Parallel and Distributed Computing and Systems (PDCS), pp. 401–406 (2000)
22. Watson, L.T., Baker, C.A.: A fully-distributed parallel global search algorithm. Eng. Comput. **18**(1/2), 155–169 (2001)
23. Zhou, J., Deng, X., Dymond, P.: A 2-D parallel convex hull algorithm with optimal communication phases. Parallel Comput. **27**(3), 243–255 (2001)